# Moose: a Collaborative and Extensible Reengineering Environment

Stéphane Ducasse, Tudor Gîrba, Michele Lanza
Software Composition Group
University of Berne – Switzerland

Serge Demeyer
Lab On Re-Engineering
University Of Antwerp – Belgium

## Abstract

Software systems are complex and difficult to analyze. Reverse engineering is a complex analysis that usually involves combining different techniques and tools. Moreover, oftentimes the existing tools are not perfectly suitable for the task, and customization of existing tools, or development of new tools is required. Moose is an extensible reengineering environment designed to provide the necessary infrastructure for tool integration. Moose centers on a language independent meta-model, and offers services like grouping, querying, navigation, and advanced tool integration mechanism.

## 1. Introduction

Reverse engineering is a complex analysis usually consisting of a combination of techniques such as: parsing the code, building a model of the code, measuring the code, visualizing, etc (Demeyer, Ducasse Tichelaar, 1999).

Sometimes, we want to complement the code information with other kinds of information like bug reports or documentation. For different techniques we need different tools, but we need these tools to collaborate and complement each other without pre-imposing the sequence of using these tools and techniques. Furthermore, real-life systems come in different shapes and sizes; therefore we need our tools to scale (Ducasse and Tichelaar, 2003).

We present Moose – an extensible and scalable reengineering environment that allows tools to collaborate. First, we show a scenario of using Moose on JBoss, a large Java open source case study. Next, we briefly introduce the most important requirements that drive Moose's design. Further we introduce the principles of Moose, and we talk about its overall architecture. We briefly

describe the underlying meta-model and the tools built on top of Moose. We show the application of our tools on a case study, and in the end we draw the conclusions and present the future work.

## 2. Moose at Work

Before detailing the internals of Moose, we first show some of the analyses that can be performed with it. As a case study we chose JBoss, an open source J2EE application server written in Java. Furthermore, to exemplify the evolutionary capabilities of Moose, we chose 10 versions starting from the beginning of 2001 until middle of 2002.

After parsing the versions with an external parser, we obtain 10 models, each model being a snapshot of one version. In Fig. 1 we show twice the main Moose window. In the left pane of the window, we show the JBoss models we have loaded. On the right side of the window we display overview measures of the selected model. The first window shows the first version we analyzed, while the second window shows the last version. For example, the first version has 632 classes and interfaces, and the last version has 3764 (the figure includes inner classes and test classes).

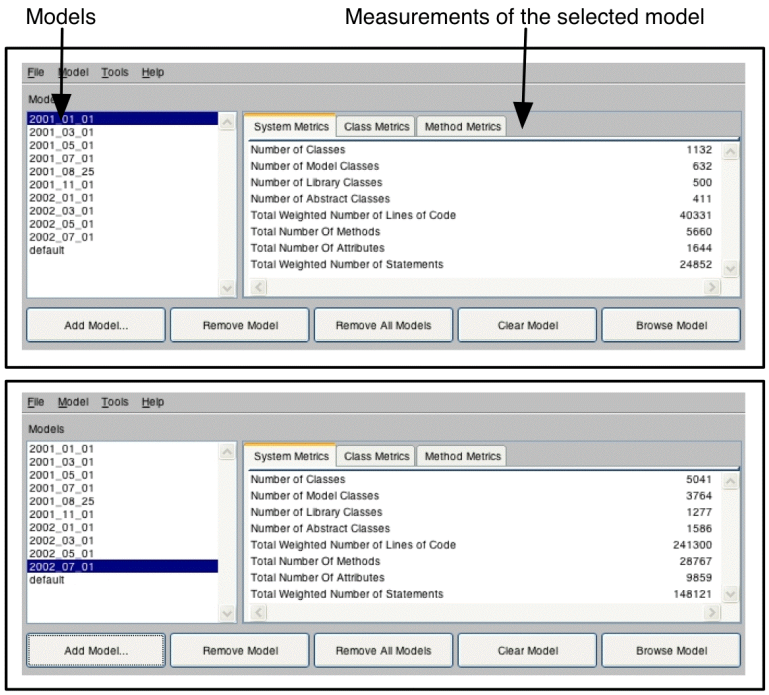


**Fig. 1 - Moode Model Manager with JBoss models**

We start the analysis by browsing the last version (see Fig. 2). In the left pane of the browser, we show the different groups of entities we have in the model: classes, methods, namespaces, invocations, accesses etc. In Fig. 2 we selected all the 5041 classes (including interfaces and inner classes) in the model and we displayed them in a table in the right pane. Furthermore, we can interact with the table and add/remove columns with different properties. In our example, we added the results of number of methods (NOM) and of lines of code (WLOC), and we ordered the classes according to the number of methods (NOM).

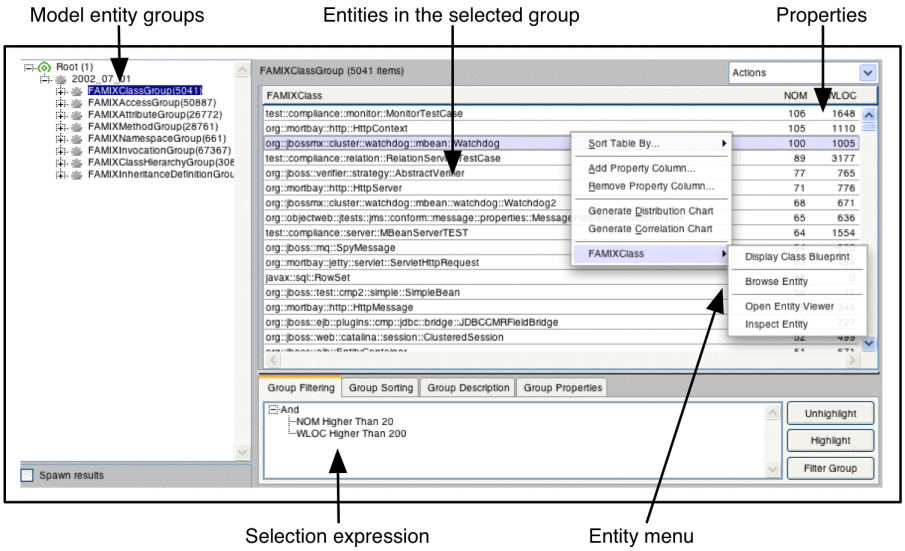


**Fig. 2 - Moose Browser on the last version of JBoss**

In the lower part of the right pane of Fig. 2 there is an editor of a visual query language, which we use to filter the entities. In our example, we would like to select the classes which have more than 20 methods and more than 200 lines of code.

From the contextual menu of the table we can choose to display the correlation between the selected properties. In Fig. 3 we display the correlation between the number of methods and the lines of code in the classes. From the shape of the chart we can detect, for example, that most of the classes have less than 50 methods and less than 1000 lines of code.

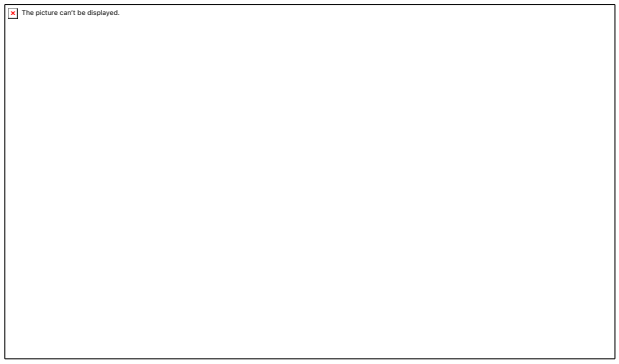


**Fig. 3 - The correlation chart between the number of methods and the lines of code of the class from the last model of JBoss.**

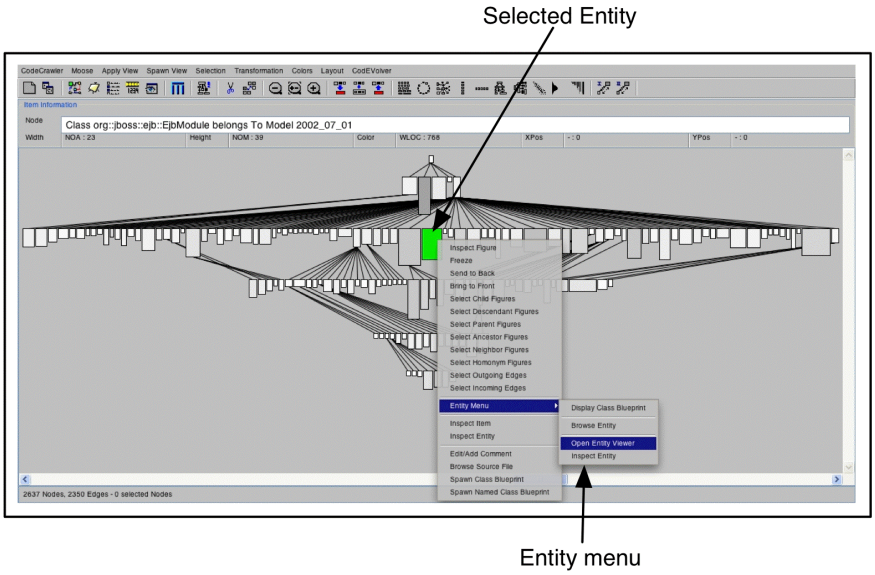


**Fig. 4 - The System Complexity view of the last model of JBoss displayed in CodeCrawler**

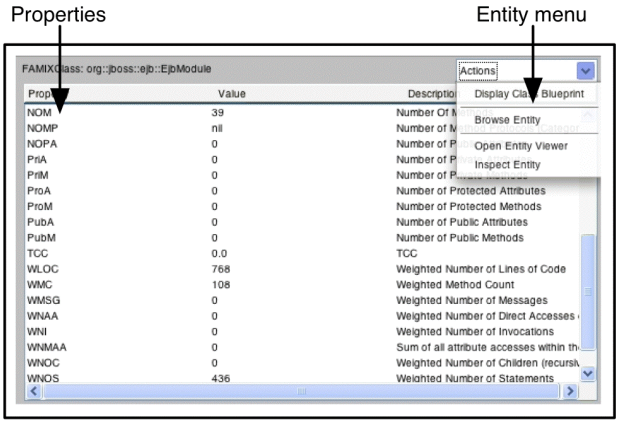


**Fig. 5. Moose entity inspector on EJBModule class from the last model of JBoss.**

To get an overview of the system, we apply a visualization called System Complexity View using CodeCrawler like in Fig. 4. This view is a polymetric view, which displays hierarchies (Lanza and Ducasse, 2003). Each class is represented as a rectangle where the height of the rectangle is given by the number of methods, the width shows the number of attributes and the color represents the lines of code of that class. With such a view we can detect exceptional classes. For example, in Fig. 4 we show the largest hierarchy of JBoss displayed in CodeCrawler. CodeCrawler is a visualization tool built on top of Moose (Lanza and Ducasse, 2004).

In Fig. 5 we show an inspector applied on the EJBModule class. The inspector shows all the properties of an entity, and it is useful for detailed analysis.

The views in CodeCrawler are interactive and we can float over the figures and obtain a contextual menu. Moreover, due to the Moose tool integration mechanisms, we can obtain the class menu, which is the same everywhere in the environment. For example, in Fig. 2, Fig. 4, and Fig. 5 we have the same contextual menu when we select a class entity. Thus, we can jump from one tool to another in a transparent way for the user.

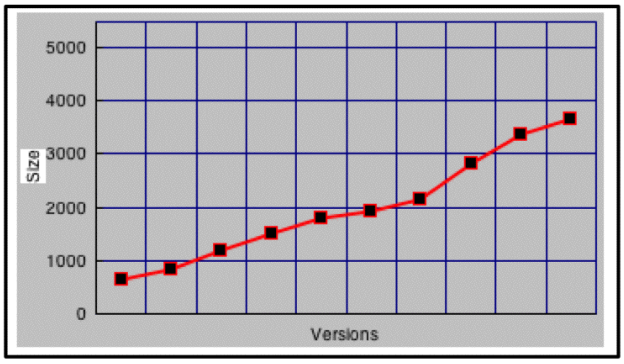


**Fig. 6 - Evolution of the number of classes in the considered history.**

Having multiple models loaded in the environment we can perform historical analyses. For example, in Fig. 6 we display the evolution chart of the number of classes in the system.

## 3. Moose Requirements

Moose is designed to meet the following requirements:

**External tools.** Reverse engineering is a complex task that can be decomposed into smaller ones that may be implemented in different tools (*e.g.,* visualization, clustering). The environment should be transparent for the user by offering mechanisms for external tools to register in order to provide a seamless integration between the tools internal and external to the environment.

**Extensible Meta Model.** Third party tools should be able to adapt the underlying meta model to their needs without breaking the interoperability. Thus, the meta model should be able to represent and manipulate entities other than the ones directly extracted from the source code *e.g.,* associations, relationships. Also, the entities in the meta-model should support annotations (*e.g.,* measurements (Ducasse Tichelaar, (2003))).

Secondly, during reverse engineering, we often need to group entities based on their commonalities. For an example, see the lower right pane in Fig. 2, where we query the Moose repository for entities of interest. Thus the environment should allow turning an arbitrary collection of entities into a separate group.

**Versioning Information.** Modern software development relies on versioning systems to store the system at different stages. The versioning information is a valuable source for reverse engineering as it can reveal, for example, which parts are changed a lot, or which ones are never changed. The environment should be able to manage multiple models at the same time for allowing the evolution observations. For instance Fig. 6 gives an overview of the number of classes in the history.

**Exploration.** The exploratory nature of reverse engineering and reengineering demands that a reengineering environment does not impose rigid sequences of activities.

The environment should be able to present the source code entities in many views, both textual and graphical, in little time. It should be possible to perform several types of actions on the views the tools provide, such as zooming, switching between different abstraction levels, deleting entities from views, grouping entities into logical clusters, etc.

The environment should as well provide a way to easily access and query the entities contained in a model. To minimize the distance between the representation of an entity and the actual entity in the source code, an environment should provide every entity with a direct link to its location in the source code.

A secondary requirement in this context is the possibility to maintain a history of all steps performed by the reengineer and preferably allow him to return to earlier states in the reengineering process: A model, in the form of entities representing the software artifacts of the target system, can be analyzed, manipulated, and browsed.

**Scalability.** As legacy systems tend to be large, an environment should be scalable in terms of the number of entities being represented, *i.e.,* at any level of granularity the environment should provide meaningful information and not incur performance penalties. An additional requirement in this context is the actual performance of such an environment. It should be possible to handle a legacy system of any size without incurring long latency times.

## 4. Moose Architecture

Moose uses a layered architecture (see Fig. 7). Information is transformed from source code into a source code model. The models are based on the FAMIX metamodel, which is described in Section 5. The information in this model, in the form of entities representing the software artifacts of the target system, can be analyzed, manipulated and used to trigger code transformations by means of refactorings. We will describe the architecture of Moose starting from the bottom.

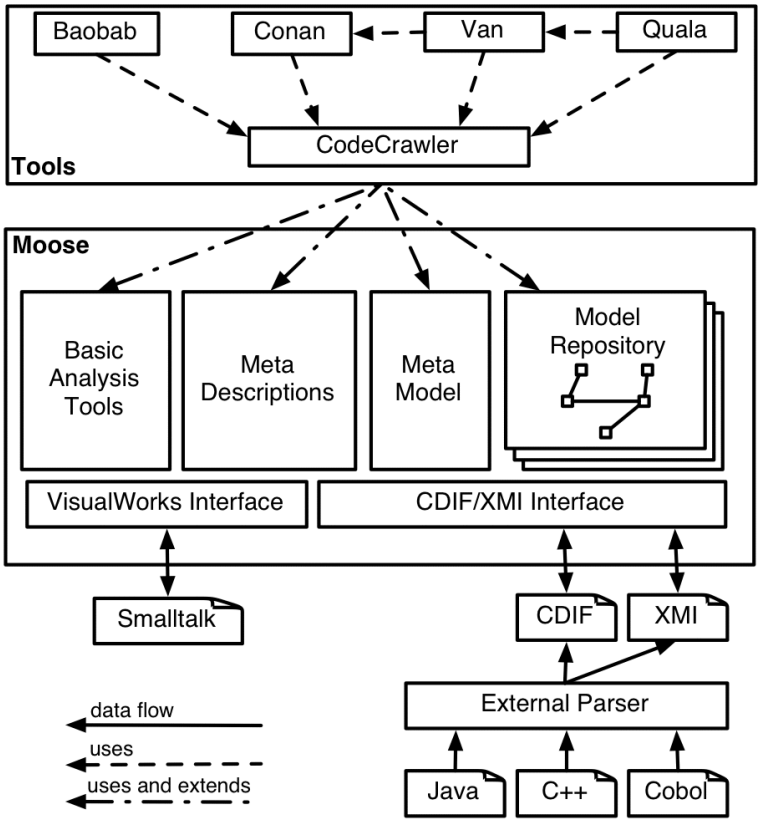


**Fig. 7 - Moose architecture.**

**Export/Import.** Moose supports multiple languages (Tichelaar, 2001). Source code can be imported into the meta-model in two different ways:

- In the case of VisualWorks Smalltalk – the language in which Moose is implemented – models can be directly extracted via the meta-model and the parser of the Smalltalk language.
- For other source languages Moose provides an import interface for CDIF and XMI files based on our FAMIX meta-model. CDIF and XMI are industrial standard interchange formats, which enable exchanging models via files or streams. Over this interface Moose uses external parsers for languages other than Smalltalk. Currently C++, Java, COBOL, and other Smalltalk dialects are supported.

**Core.** In the center of Moose is the FAMIX meta-model. The models are stored in memory. Every model contains entities representing the software artifacts of the target system. Every entity is represented by an object, which allows direct interaction and querying of entities, and consequently an easy way to query and navigate a whole model. Moose can maintain and manipulate several models in memory at the same time via a model repository.

Every entity is described by a meta-description, which is then used by the environment to display user interfaces or load/save entities. These meta-descriptions are extensible by other tools and are used by different tools. At this moment, the supported meta-descriptions are:

- *Sub entities.* Given an entity we describe the types of the sub entities that form a containment hierarchy. For example, a method is a sub entity type of a class.
- *Menu.* Every entity has a menu attached to it, and the tools can register menu actions to a particular kind of entity and this action can be triggered from everywhere in the environment. This mechanism allows for tools to collaborate with each other in a transparent way for the user.
- *Properties.* Every entity is annotated with the properties that can be computed on that entity. For example, given a class we can compute its number of methods, or denote whether the class is abstract, etc.
- *Query Expressions.* Groups of entities are first class entities, and they can be annotated with queries that can be performed to select a sub group.
- *Load/Save.* Every entity can be saved in flat file format (such as CDIF or XMI). This conversion from complex entity with relationships into a flat representation is based on the entity meta-description.

Moose provides basic tools that use the meta-descriptions:

- *Browser.* With the browser we can navigate the contents of the model.
- *Entity Inspector.* The Inspector shows all properties of a given entity.
- *Selection tool* (or query tool). It selects all entities that conform to a certain rule specified by an expression, or by a Smalltalk code. The Selection tool is part of the Browser, but it can also be used as a stand-alone tool.

**Tools.** Different tools were developed on top of Moose (S. Ducasse, M. Lanza and S. Tichelaar, 2000), such as the **Collaboration Browser** (T. Richner and S. Ducasse, 2002) and a language independent refactoring engine (S. Tichelaar, S. Ducasse, S. Demeyer and O. Nierstrasz, 2000).

Others are currently developed: **ConAn** (G. Arévalo, 2003), (G. Arévalo, S. Ducasse and O. Nierstrasz, 2004), **CodeCrawler** (Lanza and Ducasse, 2004), **Van** (T. Gîrba, S. Ducasse and M. Lanza, 2004), (D. Ratiu, S. Ducasse, T. Gîrba and R. Marinescu, 2004) and **Baobab** (S. Ducasse, M. Lanza and L. Ponisio, 2004). These tools can extend the meta-model by defining new entities and by annotating them with menus and properties.

## 5. Meta-Models

The core of Moose implements the FAMIX meta-model (Demeyer et al., 2001). FAMIX provides for a language-independent representation of object-oriented sources and contains the required information for the reengineering tasks performed by our tools (S. Tichelaar, S. Ducasse, S. Demeyer and O. Nierstrasz, 2000),.

It is language independent, because we need to work with legacy systems in different implementation languages (C++, Java, Smalltalk).

Since we cannot know in advance all information that is needed in future tools, and since for some reengineering problems tools might need to work with language-specific information (*e.g.,* to analyze include hierarchies in C++), we allow for language plug-ins that extend the model with language-specific features. Next to that, we allow tool plug-ins to extend the model to store, for instance, analysis results or layout information for graphs.

The left side of Fig. 8 shows a reduced schema of FAMIX the Moose meta-model. On the right side of the figure we show another meta-model called Hismo. Hismo is a meta-model for software evolution analysis that complements

FAMIX. The evolution of any FAMIX entity can be analyzed by its Hismo associated version/history entity.

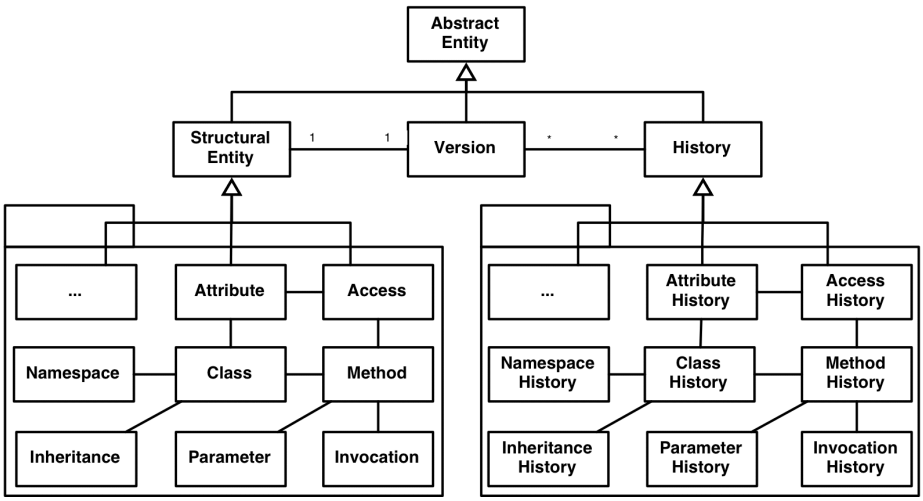


**Fig. 8 - Meta-models in Moose. On the left side we show the FAMIX meta-model, while on the right side we show Hismo, a history centric meta model which is based on FAMIX.**

To exchange model information between different tools we have adopted CDIF and XMI (Schlapbach, 2001). Both CDIF and XMI are industrial standards for transferring models created with different tools. The main reasons for adopting these formats are that firstly they are industry standards, and secondly they have a standard plain text encoding which tackles the requirements of convenient querying and human readability. Next to that the CDIF/XMI framework supports the extensibility we need to define our model and plugins. As shown in Fig. 7 we use CDIF and XMI to import FAMIX-based information about systems written in JAVA, C++ and other languages. The information is produced by external parsers such as Columbus (see www.frontendart.com), or Memoria (LOOSE Research Group, 2004).

## 6. Tools

In Fig. 9 we present a detailed schema with the tool inter-relationships. The relationship edges are annotated with the purpose. In this schema we did not represent the dependencies between these tools and Moose.

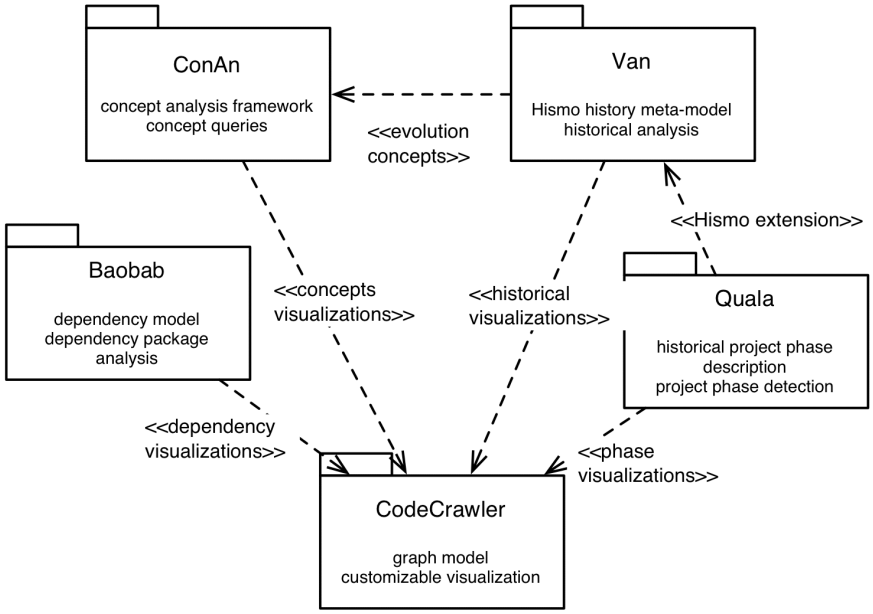


**Fig. 9 - Moose Tools and their relationships.**

**CodeCrawler**. CodeCrawler is a visualization tool implementing polymetric views (Lanza and Ducasse, 2003), which are based on a graph notion where the nodes and edges in the graph can wrap the entities in the model. For example, in Fig. 4 we see a screenshot of CodeCrawler displaying a hierarchy of JBoss. From the picture we see that from CodeCrawler we can access the menu related to the node we select because we have access to the entity.

Although CodeCrawler was first developed as a visualization tool for software systems, in its latest implementation it turned into a general-purpose visualization tool, which can accommodate different needs. For example, in Fig. 9 it is shown that all the tools use CodeCrawler for different visualizations.

**Baobab**. Baobab is a tool to understand dependencies between modules (S. Ducasse, M. Lanza and L. Ponisio, 2004). It extends FAMIX with the notion of dependency between different parts of the system and provides various measurements for these dependencies. It uses CodeCrawler for showing the types of modules in the system (*e.g.,* provider modules, client modules).

**ConAn**. ConAn is a concept analysis tool and manipulates concepts as first class entities. Its target is to detect different kinds of patterns in the model based on combining elements and properties.

ConAn uses CodeCrawler for visualization purposes and supports analyses like:

- *X-Ray views for understanding the internal of classes*. The X-Rays aim at describing the details of classes. For example, they can show how state is used in a class and detect if the class attributes are used in chunks (G. Arévalo, S. Ducasse and O. Nierstrasz, 2004).
- *Identification of recurring code patterns*. ConAn can detect design patterns (e.g., Composite design pattern) or other kind of patterns that frequently occur in the code (e.g., Star Classes are those classes which are used by a lot of other classes) (G. Arévalo, Frank Buchli and Oscar Nierstrasz, 2004).
- *Views for hierarchy understanding*. ConAn facilitates the understanding of class hierarchies as a whole by detecting, for example, methods which reuse the behavior in the super class, or methods which access directly the state in the super class (G. Arévalo, 2003).

**Van**. Van is a tool for analyzing the evolution of systems. At its core, it defines the Hismo meta-model which is based on the notion of history (see Fig. 7). Hismo is independent from FAMIX, but it works closely with it. Van offers different analyses.

- *Changes characterization based on historical measurements*. Van defines different measurements for summarizing the evolution of entities that allow for comparison of different evolutions (T. Gîrba, S. Ducasse and M. Lanza, 2004).
- *Improved design flaws detection*. Van uses historical information to improve the detection of design flaws. For example, a if a GodClass is detected in the latest version, but in the same time it was changed very seldom in its history we say it is a HarmlessGodClass because it was not a maintainability problem in the past (D. Ratiu, S. Ducasse, T. Gîrba and R. Marinescu, 2004).
- *Evolution visualization*. Van uses CodeCrawler to produce different visualization of the evolution of parts of the system. For example,

we provide a visualization to understand different evolution patterns for entire class hierarchies (T. Gîrba and M. Lanza, 2004).

- *Hidden dependencies detection based on change information*. Van uses ConAn for detecting concepts like parallel inheritance based on change information (T. Gîrba, S. Ducasse, R. Marinescu and D. Ratiu, 2004).
- *Past refactorings detection*. Having the historical information about the code, Van can detect past refactorings like renamings, movings, splitting.

**Quala**. Quala is built on top of Van and is a tool used to describe different phases of the evolution of the system. It extends the Hismo meta-model with the concept of phase as a first class entity, which is a sub-history which complies with a certain rule. For example, Quala can detect growing phases, shrinking phases, refactoring phases. These phases are used to describe the evolution of entities.

## 7. Industrial Validation

Moose and the tools built on top of it have been used to reverse engineer industrial systems several times. Due to non-disclosure agreements with the industrial partners we cannot provide detailed descriptions of our experiences, but limit ourselves to provide a list of case studies (industrial and non-industrial) that we have performed.

| System | Language | Lines of Code | Classes |
|---|---|---|---|
| Z (Network Switch) | C++ | 1'200'000 | ~2300 |
| Y (Network Switch) | C++/Java | 140'000 | ~400 |
| X (Multimedia) | Smalltalk | 600'000 | ~2500 |
| W (Payroll) | COBOL | 40'000 | - |
| SORTIE (Forest Management) | C/C++ | 28'000 | ~70 |
| Duploc (Research Prototype) | Smalltalk | 32'000 | ~230 |
| Jun (Multimedia and 3D Framework) | Smalltalk | 135'000 | ~700 |
| Squeak (Multimedia Environment) | Smalltalk | 260'000 | ~1800 |
| JBoss (Application Server) | Java | 300'000 | ~4900 |
| V (Logistics) | C++ | 120'000 | ~300 |

**Tab. 1 – A Selection of the Case Studies performed with Moose.**

In Tab. 1 we see that the systems were written in different programming languages and have sizes quite different from each other. As far as scaleability

concerns, we point out that we never had more than one week to reverse engineer the systems. Despite these narrow time constraints imposed on us, we always have been able to extract information which was deemed relevant by the actual maintainers of the software system.

## 8. Conclusions and Future Work

Reverse engineering is a complex task and requires combining different techniques and tools. We presented Moose, a reengineering environment designed to be:

- extensible, to support unexpected needs,
- exploratory, to allow for flexible sequence of actions, and
- scalable, to cope with large systems.

In this chapter we briefly showed Moose at work when analyzing 10 versions of JBoss, a large open source J2EE application server. We showed how we browse the model, how we use measurements, how we visualize the model and how we combine different tools and techniques.

On top of Moose several tools were built, each of them having its own focus: clustering, visualization, concept analysis, and version analysis. We showed how they extend Moose and how collaborate with each other using meta descriptions.

## Moose Availability

Moose is completely implemented in Smalltalk under the BSD license: it is *free and open source software*. Moose runs on every major platform (Windows, Mac OS, Linux, Unix). Moose is freely available for download. The current webpage of Moose is located at:

http://www.iam.unibe.ch/~scg/Research/Moose/.

Moreover, Moose is also available as free goodie on the VisualWorks Smalltalk CD, a professional, commercial development environment developed and sold by the company Cincom which however also exists in a non-commercial version freely available for download at:

http://www.cincomsmalltalk.com/

## Bibliography


G. Arévalo (2003), *"Understanding Behavioral Dependencies in Class Hierarchies using Concept Analysis," Proceedings of LMO 2003: Langages et Modeles à Objets*, Hermes, Paris, January 2003, pp. 47-59.

G. Arévalo, S. Ducasse and O. Nierstrasz (2004), *"X-Ray Views: Understanding the Internals of Classes,"* Proceedings of ASE 2003, IEEE Computer Society, October 2003, pp. 267—270

G. Arévalo, Frank Buchli and Oscar Nierstrasz (2004), "*Detecting Implicit Collaboration Patterns*," Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004), IEEE Computer Society Press, November 2004

S. Demeyer, S. Ducasse, and S. Tichelaar (1999). "*Why Unified is not Universal. UML – shortcomings for Coping with Round-trip Engineering*" Proceedings of UML'99, LNCS 1723, 1999.

S. Demeyer, S. Tichelaar, and S. Ducasse (2001). "FAMIX 2.1 — the FAMOOS information exchange model". Technical report, University of Bern.

S. Ducasse, M. Lanza and S. Tichelaar (2000), *"Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems,"* Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000), June 2000.

S. Ducasse and S. Tichelaar (2004), *Dimensions of Reengineering Environment Infrastructures*, In International Journal on Software Maintenance: Research and Practice, October, Volume 15, pp. 345--373, 2003

S. Ducasse, M. Lanza and L. Ponisio (2004), *"A Top-Down Program Comprehension Strategy for Packages,"* Under Submission, 2004

T. Gîrba, S. Ducasse and M. Lanza (2004), **"***Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes*," 20th International Conference on Software Maintenance (ICSM 2004).

T. Gîrba and M. Lanza (2004), *"Visualizing and Characterizing the Evolution of Class Hierarchies***,"** Fifth International Workshop on Object-Oriented Reengineering (WOOR 2004), 2004

T. Gîrba, Stéphane Ducasse, Radu Marinescu and Daniel Ratiu, *"Identifying Entities That Change Together,"* Ninth IEEE Workshop on Empirical Studies of Software Maintenance (WESS 2004), 2004

M. Lanza and S. Ducasse (2003), "Polymetric Views — A Lightweight Visual Approach to Reverse Engineering". *IEEE Transactions on Software Engineering*, 29(9): 782–795, Sept. 2003.

M. Lanza and S. Ducasse (2004), "CodeCrawler – An Extensible and Language Independent 2D and 3D Software Visualization Tool", this book.

D. Ratiu, S. Ducasse, T. Gîrba and R. Marinescu (2004), *"Using History Information to Improve Design Flaws Detection,"* Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), pp. 223—232.

T. Richner (2002), *"Recovering Behavioral Design Views: a Query-Based Aproach,"* Ph.D. thesis, University of Berne, May 2002

T. Richner and S. Ducasse (2002), *"Using Dynamic Information for the Iterative Recovery of Collaborations and Roles,"* Proceedings of ICSM '2002 (International Conference on Software Maintenance).

A. Schlapbach (2001). "Generic XMI support for the MOOSE reengineering environment" Technical Report, University of Bern.

LOOSE Research Group (2004), "Memoria". http://www.loose.utt.ro/.

S. Tichelaar, *"Modeling Object-Oriented Software for Reverse Engineering and Refactoring,"* Ph.D. thesis, University of Berne, 2001

S. Tichelaar, S. Ducasse, S. Demeyer and O. Nierstrasz (2000), **"***A Meta-model for Language-Independent Refactoring,"* Proceedings ISPSE 2000 (International Conference on Software Evolution), IEEE, 2000, pp. 157-167.